20 algorithms in JavaScript

This is about all 20 algorithms, the shortest list of algorithms in JavaScript that I could find, from the following tutorial

https://www.geeksforgeeks.org/learn-algorithms-with-javascript-tutorial/

but none of the algorithms and none of the explanations, are from that tutorial. The algorithms are widely known algorithms, taken from the following project, converted to JavaScript and modified

https://github.com/TheAlgorithms/Java .

Algorithms are machines. As all of these algorithms are implemented in some libraries, there may not be a need today to write one oneself. **Their main importance today is to know how to write code. There are not many theories about writing code in computer science, thus what concerns all the rest, all there is are algorithms**.

There is a reference to the pdf used in this video in the description, so you can read it and also run the code directly by clicking on the links, as long as the code remains in OneCompiler.

© 2024, Tarvo Korrovits
This work is licensed under https://creativecommons.org/licenses/by/4.0/

Table of Contents

Linear Search

https://onecompiler.com/javascript/426gxg4gm

Linear Search is the simplest search, iterating through the array, and when value is found, returning the index. **There is a function for linear search, so there is really no need to write a loop**.

```
const array = [10, 20, 30, 40, 50];
console.log(array.indexOf(40));
console.log(array.indexOf(4));
```

Binary Search

https://onecompiler.com/javascript/3zyxp2rgg

Binary Search assumes that the array is sorted. This is an iterative implementation. **In every iteration it looks at the left and right half of the segment**. We search in segment, that starts with l and ends with r. We do it by calculating mid, the middle index of the segment. If the value is at that index, then we found it, otherwise we will look into segment to the right or to the left of it, depending on whether the value is greater or less than the element at that index.

```
const binarySearch = (array, value) => {
	let l = 0
	let r = array.length - 1;
	while (l <= r) {
		const mid = l + r >> 1;
		if (array[mid] == value) return true;
		if (array[mid] < value)
			l = mid + 1;
		else
			r = mid - 1;
	}
	return false;
}

const array = [1, 4, 7, 9, 11, 12];
console.log(binarySearch(array, 4));
console.log(binarySearch(array, 5));
```

Bubble Sort

https://onecompiler.com/javascript/3zyy53x8e

Bubble Sort goes repeatedly through the array, each time swapping the elements that follow each other, that are in the wrong order, in this case ascending order. As each time when going through the array, the largest value will be at the end, every next time one more element at the end can be omitted.

```
const swap = (a, i, j) => [a[i], a[j]] = [a[j], a[i]];

const bblSort = array => {
        for (let i = 0; i < array.length; i++)
                for (let j = 0; j < array.length - i; j++)
                        if (array[j] > array[j + 1])
                                swap(array, j, j + 1);
}

const array = [6, 3, 7, 4, 8];
console.log("Before sorting:");
console.log(array);
bblSort(array);
console.log("After sorting:");
console.log(array);
```

Insertion Sort

https://onecompiler.com/javascript/426gyeza5

Insertion Sort goes through the array, and in every iteration moves the value of the current element back, where it belongs by the sorting order. This works because after every iteration, the array up to the current element is already sorted.

```
const insertionSort = array => {
	for (let i = 0; i < array.length; i++) {
		let j;
		const current = array[i];
		for (j = i - 1; j >= 0 && array[j] > current; j--)
			array[j + 1] = array[j];
		array[j + 1] = current;
	}
}

const array = [16, 9, 12, 4, 7];
console.log("Before sorting:");
console.log(array);
insertionSort(array);
console.log("After sorting;");
console.log(array);
```

Selection Sort

https://onecompiler.com/javascript/426gytrtu

Selection Sort goes through the array, and for every element it finds an element after it, that is minimal, and then swaps the values of these two elements. Thus in every iteration, the array before the current index is already sorted.

```
const swap = (a, i, j) => [a[i], a[j]] = [a[j], a[i]];

const selectionSort = array => {
	for (let i = 0; i < array.length; i++) {
		let minIndex = i;
		for (let j = Number(i) + 1; j < array.length; j++)
			if (array[j] < array[minIndex])
				minIndex = j;
		swap(array, minIndex, i);
	}
}

const array = [45, 22, 14, 19, 10];
console.log("Before sorting:");
console.log(array);
selectionSort(array);
console.log("After sorting:");
console.log(array);
```

Towers of Hanoi

https://onecompiler.com/javascript/3zz2r2mez

Towers of Hanoi is a recursive algorithm. Recursive algorithms are top down, they go down to the base case. **Here the base case is the number n equals 0, from which it starts to return. Every time the disk of some number will be moved from the source rod to the auxiliary rod, and then from the auxiliary rod to the destination rod, using the remaining rod as the auxiliary rod**. The top case is moving the bottom disk 3 from rod A to rod C, using rod B as an auxiliary rod.

```
const hanoi = (n, from, to, aux) => {
        if (!n) return;
        hanoi (n - 1, from, aux, to);
        console.log("Disk " + n + " from " + from + " to " + to);
        hanoi(n - 1, aux, to, from);
}

hanoi(3, "A", "C", "B");
```

Depth First Search

https://onecompiler.com/javascript/3zz473twe

Depth First Search traverses the graph recursively, for each node traversing the adjacent nodes. Here the graph is represented as adjacency list, an array with a list of indexes of the adjacent nodes, for every node. There is a separate array showing whether the nodes have been visited, necessary for graphs that may have cycles in them. The implementation here is only a traversal, as it doesn't search anything.

```
const dfs = (adjList, visited, v) => {
	console.log(v);
	visited[v] = true;
	for (let e of adjList[v])
		if (!visited[e])
			dfs(adjList, visited, e);
}

const adjList = [
	[1, 2],
	[2],
	[0, 3],
	[3]
];
const visited = Array.from({length: adjList.length}, e => false);
dfs(adjList, visited, 0);
```

Fibonacci Numbers

https://onecompiler.com/javascript/3zz3678c8

Fibonacci numbers are numbers in the sequence, where every number is sum of the two preceding numbers. The implementation here is recursive, with the base case n <= 1 .

```
const fibonacci = n => {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(14));
```

Sudoku

https://onecompiler.com/javascript/426gzgmj7

Sudoku algorithm uses backtracking, that is trying solutions, it cannot find all solutions. IsSafe() checks if a number can be placed, that is there is not the same number in the same row, column, or the 3x3 box. **The base case is when the end of board was reached**. When reaching the end of row, it goes to the next row. **If the cell contains number, next cell will be checked. Otherwise it is checked if a number can be placed**. If a check succeeds, next cell will be checked, otherwise the cell will be made 0 again. If no number can be placed, then it cannot be solved.

```
const isSafe = (board, row, col, num) => {
	for (let i = 0; i < board.length; i++)
		if (board[row][i] == num || board[i][col] == num)
		  return false;
	for (let i = 0; i < 3; i++)
		for (let j = 0; j < 3; j++)
			if (board[row - row % 3 + i][col - col % 3 + j] == num)
				return false;
	board[row][col] = num;
	return true;
}

const solveSudoku = (board, row, col) => {
	if (row == board.length - 1 && col == board.length) return true;
	if (col == board.length) [row, col] = [row + 1, 0];
	if (board[row][col] != 0) return solveSudoku(board, row, col + 1);
	for (let num = 1; num <= 9; num++) {
		if (isSafe(board, row, col, num))
			if (solveSudoku(board, row, col + 1))
				return true;
		board[row][col] = 0;
	}
	return false;
}

const board = [
	[3, 0, 6, 5, 0, 8, 4, 0, 0],
	[5, 2, 0, 0, 0, 0, 0, 0, 0],
	[0, 8, 7, 0, 0, 0, 0, 3, 1],
	[0, 0, 3, 0, 1, 0, 0, 8, 0],
	[9, 0, 0, 8, 6, 3, 0, 0, 5],
	[0, 5, 0, 0, 9, 0, 6, 0, 0],
	[1, 3, 0, 0, 0, 0, 2, 5, 0],
	[0, 0, 0, 0, 0, 0, 0, 7, 4],
	[0, 0, 5, 2, 0, 6, 3, 0, 0]
];
solveSudoku(board, 0, 0);
for (let i = 0; i < board.length; i++) {
	let str = "";
	for (let j in board) str += board[i][j] + "  ";
	console.log(str);
}
```

M Coloring Problem

https://onecompiler.com/javascript/426hrk5xx

M Coloring algorithm is also a backtracking algorithm. IsSafe() checks all pairs of vertices. If there is an edge for any pair, and it's vertices are colored with the same color, it returns false. **The base case is when the end of graph was reached**, checking whether all graph was colored correctly. **The recursive case is checking whether a color can be assigned to the vertex**, each time the next vertex is checked, and if that fails, the color will be made 0 again. If no color can be assigned, then it cannot be solved.

```
const isSafe = (adjList, colors) => {
	for (let i = 0; i < adjList.length; i++)
		for (let j = i + 1; j < adjList.length; j++)
			if (adjList[i].includes(j) && colors[j] == colors[i])
				return false;
	return true;
}

const mColoring = (adjList, colors, m, v) => {
	if (v == adjList.length) return isSafe(adjList, colors);
	for (let i = 1; i <= m; i++) {
		colors[v] = i;
		if (mColoring(adjList, colors, m, v + 1)) return true;
		colors[v] = 0;
	}
	return false;
}

const adjList = [
	[1, 2, 3],
	[0, 2],
	[0, 1, 3],
	[0, 2]
];
const colors = Array.from({length: adjList.length}, e => 0);
mColoring(adjList, colors, 3, 0);
for (let e of colors) console.log(e);
```

N Queens Problem

https://onecompiler.com/javascript/426hsbmk6

N Queens algorithm is a backtracking algorithm. The board is an array of the row numbers of the queens. The base case is when the end of board was reached, and there the content of the board is added to the solutions array. **The recursive case is trying to place a queen to the current column**. IsSafe() checks for each row if the row is not equal to the row of any previously placed queens, and the difference of rows is not equal to the difference of columns. If true, placing queen to the next column will follow. As there is no return after the recursive call, several solutions can be found.

```
const isSafe = (board, row, col) => {
	if (!board.length) return false;
	for (let i = 0; i < col; i++) {
		const diff = Math.abs(board[i] - row);
		if (diff == 0 || diff == col - i) return false;
	}
	return true;
}

const solveNQ = (n, sols, board, col) => {
	if (n && col == n) {
		const sol = [];
		for (let i = 0; i < board.length; i++) {
			let str = "";
			for (let j in board) str += j == board[i] ? "Q " : ". ";
			sol.push(str);
		}
		sols.push(sol);
		return;
	}
	for (let row = 0; row < n; row++) {
		board[col] = row;
		if (isSafe(board, row, col)) solveNQ(n, sols, board, col + 1);
	}
	return;
}

for (let n = 2; n <= 6; n++) {
	const sols = [];
	solveNQ(n, sols, [], 0);
	if (!sols.length) {
		console.log("No way to place " + n + " queens");
		console.log();
		continue;
	}
	console.log("Solutions for " + n + " queens:");
	console.log();
	for (let sol of sols) {
		for (let e of sol) console.log(e);
		console.log();
	}
}
```

Catalan Numbers

https://onecompiler.com/javascript/3zz942ss4

Catalan Numbers algorithm is a typical recursive algorithm, with the base case n <= 1 , and the recursive case a loop, that calculates Catalan number from previous Catalan numbers. Catalan numbers are used in combinatorics.

```
const catalan = n => {
        if (n <= 1) return 1;
        let sum = 0;
        for(let i = 0; i < n; i++) sum += catalan(i) * catalan(n - i - 1);
        return sum;
}

for (let i = 0; i < 10; i++) console.log(catalan(i));
```

Binomial Coefficient

https://onecompiler.com/javascript/3zz95kgmc

Binomial Coefficient algorithm is also a recursive algorithm, with the base case `k > n` and `k == 0 || k == n` , the recursive case calculates binomial coefficient from previous binomial coefficients. Binomial coefficient is used for expanding `(x + y) ** n` into a sum of terms `a * x ** b * y ** c` , where `b + c == n` and a is `binomial(n, b)` , that is equal to `binomial(n, c)` . In combinatorics, binomial coefficient gives the number of ways in which k objects can be chosen from the set of n objects.

```
const binomial = (n, k) => {
       if (k > n) return 0;
       if (k == 0 || k == n) return 1;
       return binomial(n - 1, k - 1) + binomial(n - 1, k);
}

console.log(binomial(5, 2));
```

Subset Sum

https://onecompiler.com/javascript/3zzbfjv78

Subset Sum algorithm is a bottom up dynamic programming algorithm that uses memoizing. Bottom up algorithms are iterative. The base case is setting the upper row to true. Array isSum has a row for every sum. When isSum[i][j] is true, this means that there is a subset in arr[0 … j - 1] , the sum of which is equal to i. The array is iterated, and in every iteration it is iterated through columns. The value from previous column is copied, as longer set also contains the subsets that were found before. If i is greater than the current array element (j – 1, because array starts from 0), then it will be found whether in the previous column the value corresponding to that difference, is true, that is the set with that sum was already found. Finally, if a set with the sum was found, the last column of the last row will contain true.

```
const subsetSum = (arr, sum) => {
        const n = arr.length;
        const isSum = Array.from({length: sum + 1}, e =>
          Array.from({length: n + 1}, e => false));
        for (let j = 0; j <= n; j++) isSum[0][j] = true;
        for (let i = 1; i <= sum; i++)
                for (let j = 1; j <= n; j++) {
                        isSum[i][j] = isSum[i][j - 1];
                        if (i >= arr[j - 1])
                          isSum[i][j] ||= isSum[i - arr[j - 1]][j - 1];
                }
        return isSum[sum][n];
}

const arr = [50, 4, 10, 15, 34];
console.log(subsetSum(arr, 64));
console.log(subsetSum(arr, 99));
console.log(subsetSum(arr, 5));
console.log(subsetSum(arr, 66));
```

Prime Numbers

https://onecompiler.com/javascript/3zzegmfhh

Prime Numbers algorithm uses sieve of Eratosthenes, that is in a way also a form of memoizing. The upper limit is square root of n, because every not prime number has at least one factor that is less than its square root. For the same reason, we start sieving from i * i, as all not prime numbers before are already marked off, because they have a factor less than i. We create an array for numbers 0 to n, with all elements true. We iterate through it, and if the element at i is true, that is prime, we iterate to the end of array with step i, marking elements false.

```
const eratosthenes = n => {
	const array = Array.from({length: n + 1}, e => true);
	const upper = Math.floor(Math.sqrt(n));
	for (let i = 2; i <= upper; i++)
		if (array[i])
			for (var j = i * i; j <= n; j += i)
				array[j] = false;
	return array.reduce((t, v, i) => v && i > 1 ? t.concat(i) : t, []);
};

console.log(eratosthenes(11));
```

Lowest Common Multiplier

https://onecompiler.com/javascript/3zzew4h29

Lowest Common Multiplier algorithm goes through the array, and for each element it calculates new multiplier, by multiplying the previous multiplier m by the value of the element v, and dividing it by the greatest common divisor of these two numbers. When doing so, the multiplier can be divided by both. The greatest common divisor is found in the same way as in the Euclidean algorithm.

```
const gcd = (a, b) => !b ? a : gcd(b, a % b);

const lcm = arr => arr.reduce((m, v) => v * m / gcd(v, m), arr[0]);

console.log(lcm([2, 7, 3, 9, 4]));
```

Euclidean Algorithm

https://onecompiler.com/javascript/3zzhz6n6n

Euclidean Algorithm repeatedly replaces one number with the remainder of dividing that number by the other, because then the greatest common divisor doesn't change. The base case is when the remainder is 0, then the other number is the greatest common divisor. The algorithm also calculates the Bezout coefficients xy == [x, y] , so that a * x + b * y == gcd(a, b) every time after returning from the recursive function. How they are calculated comes from the derivation a * x + b * y == b % a * x0 + a * y0 == a * (y0 – b / a * x0) + b * x0 .

```
const gcd = (a, b, xy) => {
        if (a < 0 || b < 0) return null;
        if (!a) {
                [xy[0], xy[1]] = [0, 1];
                return b;
        }
        const g = gcd(b % a, a, xy);
        const [x, y] = [xy[0], xy[1]];
        [xy[0], xy[1]] = [y - Math.floor(b / a) * x, x];
        return g;
}

const xy = [0, 0];
console.log(gcd(35, 15, xy), xy);
```

Count Set Bits

https://onecompiler.com/javascript/3zzjcfr6u

Count Set Bits algorithm counts bits that are 1 in the binary number. n & (n – 1) always removes one 1, as subtracting one from the number that ends with 1 and a number of zeroes, results in the number where these zeroes will be ones, and the leading 1 will be zero. Bitwise and of that and the original number, results in the leading 1 removed, and zeroes after that remaining zero. The number will end with any 1 there is before, followed by zeroes. Like if the number ends with 1000, after subtracting one it ends with 0111, and after and, it ends with 0000. This requires only one iteration for each 1. Speed is important, because the algorithm can be used for calculating checksums for large amounts of data.

```
const countSetBits = n => {
        for (var cnt = 0; n; cnt++) n &= n - 1;
        return cnt;
}

console.log(countSetBits(9));
```

Add Bit Strings

https://onecompiler.com/javascript/3zzmhn2ga

Add Bit Strings algorithm adds binary numbers represented as strings. It makes the lengths equal, by adding a number of zeroes to the smaller string. Then it goes through the strings backwards, each time calculating the bit values of the current characters. From these and carry, the result bit is calculated. We exclusive or the bits, that is 1 if only one of them is 1. Then we exclusive or the first part and carry, that is also 1 if only one of these is 1. The carry is 1 if both bits are 1, and also if either of the bits is 1 and the previous carry is 1. If carry is 1 after the loop, "1" will be added to the beginning of the result.

```
const addBinary = (s1, s2) => {
        const len = s1.length > s2.length ? s1.length : s2.length;
        s1 = "0".repeat(len - s1.length) + s1;
        s2 = "0".repeat(len - s2.length) + s2;
        let carry = 0;
        let result = "";
        for (let i = len - 1; i >= 0; i--) {
                const first = s1[i] - "0";
                const second = s2[i] - "0";
                const resultBit = first ^ second ^ carry;
                result = String.fromCharCode(resultBit + 48) + result;
                carry = first & second | first & carry | second & carry;
        }
        if (carry) result = "1" + result;
        return result;
}

console.log(addBinary("1100011", "10"));
```

Parity

https://onecompiler.com/javascript/3zzjf8kce

Parity algorithm is the same as the Count Set Bits algorithm, except that instead of counting the ones in the binary form of the number, it finds whether there is even or odd number of ones.

```
const parity = n => {
	for (var p = true; n; p = !p) n &= n - 1;
	return p;
}

console.log(parity(7) ? "even": "odd");
```